



CarnegieMellon
Software Engineering Institute

Architecture-Based Development

Len Bass
Rick Kazman

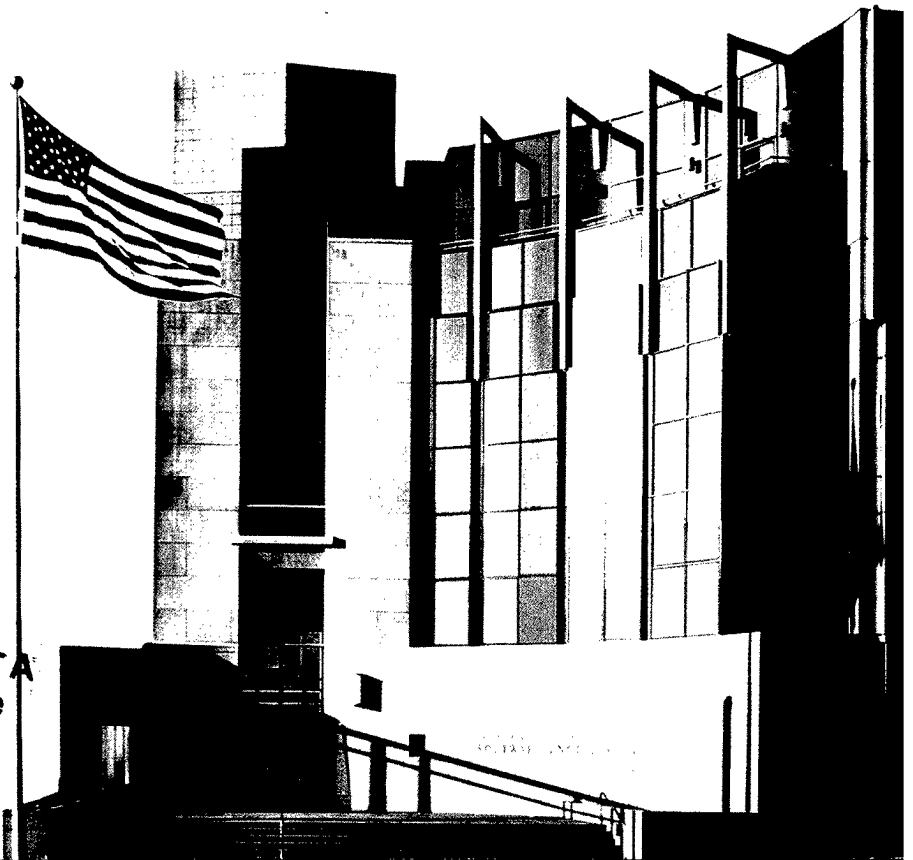
April 1999

19990728 004

TECHNICAL REPORT
CMU/SEI-99-TR-007
ESC-TR-99-007

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

DTIC QUALITY INSPECTED 4



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.



**Carnegie Mellon
Software Engineering Institute**

Pittsburgh, PA 15213-3890

Architecture-Based Development

CMU/SEI-99-TR-007
ESC-TR-99-007

Len Bass
Rick Kazman

April 1999

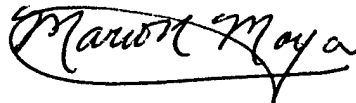
Product Line Systems

Unlimited distribution subject to the copyright.

This report was prepared for the
SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Mario Moya, Maj, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright © 1999 by Carnegie Mellon University.

Requests for permission to reproduce this document or to prepare derivative works of this document should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

This document is available through Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: <http://www.asset.com> / e-mail: sei@asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800-225-3842.

Table of Contents

1	Introduction	1
2	Requirements	3
2.1	Architectural Requirements	3
2.2	Quality Scenarios	4
2.2.1	Abstract Scenarios	6
2.2.2	Quality-Specific Scenarios	7
2.3	Validation	7
3	Design the Architecture	9
3.1	Architectural Structures and Views	10
3.2	A Development Process	11
3.3	Validation	13
4	Document the Architecture	15
5	Analyze the Architecture	19
5.1	Architectural Reviews	19
5.1.1	Participants	20
5.1.2	Review Techniques	20
5.2	Architecture Tradeoff Analysis Method (ATAM)	21
5.3	The Steps of the ATAM	22
5.3.1	Day 1 Activities	23
5.3.2	Day 2 Activities	25
5.3.3	Day 3 Activities	27
6	Realize the Architecture	29
7	Maintain the Architecture	31
8	Conclusions	33
	References	35

List of Figures

Figure 1-1	Steps of the Architecture-Based Development Process	2
Figure 2-1	Eliciting the Architectural Requirements	3
Figure 2-2	A Scenario-Elicitation Matrix	5
Figure 3-1	Architecture Design	9
Figure 4-1	Architecture Documentation	15
Figure 5-1	Architecture Analysis	19
Figure 5-2	The Activities of ATAM and Their Relative Importance Over Time	22
Figure 5-3	Dependencies Among ATAM Outputs	23

List of Tables

Table 3-1	Views Useful for Different Qualities	11
-----------	--------------------------------------	----

Abstract

This report presents a description of architecture-centric system development. In an architecture-centric process, a set of architecture requirements is developed in addition to functional requirements. This report describes the source of these architecture requirements and how they are elaborated into a design. In addition to design, the documentation, evaluation, realization, and maintenance of an architecture are also described.

1 Introduction

The development of a software architecture is a critical step in the development of large software-intensive systems. A software architecture is fundamental for the development of software produce lines where multiple systems with different functionality are created from the same basic architecture. Even with this emphasis on architecture, the process of defining and maintaining an architecture remains vague. For example, there is an architecture-development process from the object-oriented community based on the analysis of a collection of “use cases” [Jacobson 92]. This process is designed to identify the objects that exist within a system and their interaction. However, it does not provide a clear method for defining an architecture. There is another process from the architecture community that proposes basing an architecture on architectural styles with known properties without discussing how to move beyond the information gathered from the styles [Shaw 96]. While neither description of the architecture development process is incorrect, they are not totally correct either. Furthermore, neither of these processes accurately reflect the process that architects actually use to design architectures. In this report, we present a process for deriving an architecture that comes from our work with large systems and, specifically, with the architects of these systems.

Our experience is based on working closely with architects who have designed systems for large civilian corporations and military contractors. All of the systems that we examined were larger than 100 KSLOC (thousand source lines of code). Although none of the architects has used exactly the process that we describe in this report, it is an abstraction to which we believe all of them would subscribe.

The process described in this report includes the following steps:

1. Elicit the architectural requirements.
2. Design the architecture.
3. Document the architecture.
4. Analyze the architecture.
5. Realize the architecture.
6. Maintain the architecture.

Each of these steps includes

- inputs, including the means of collecting this information
- constructive activities
- validation activities
- outputs

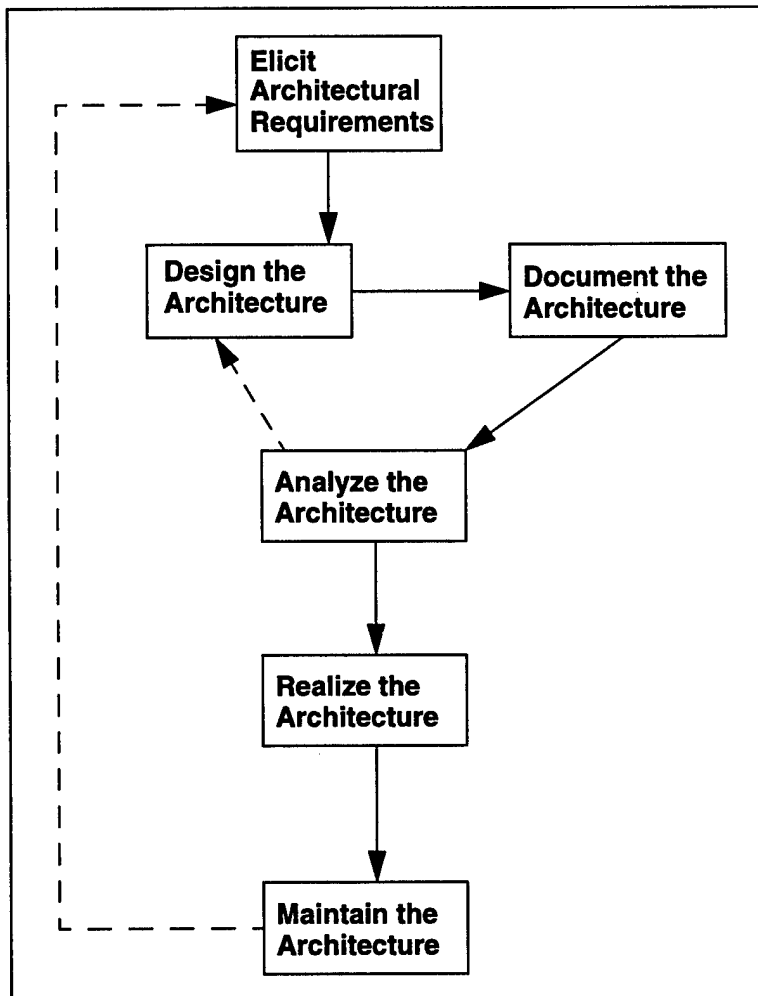


Figure 1-1: Steps of the Architecture-Based Development Process

In this report, we describe the steps of this process as show in Figure 1-1. The steps of design, documentation, and analysis form an iterative portion of the process. Once an acceptable architecture is achieved, then it is realized and then must be maintained. In Sections 2-7, we provide a notional picture of each step, its iteration, and the artifacts and people that provide input into the steps.

2 Requirements

Figure 2-1 provides an overview of the requirements elicitation process. It shows that architectural requirements are created by the developing organization and are influenced by the technical environment and the architect's own experience. We will discuss the inputs into the process in subsequent sections. There are three outputs from the process: an enumeration of functional requirements made concrete by use cases, an enumeration of specific architectural requirements, and an enumeration of a collection of quality scenarios that provide concrete tests for the architectural requirements.

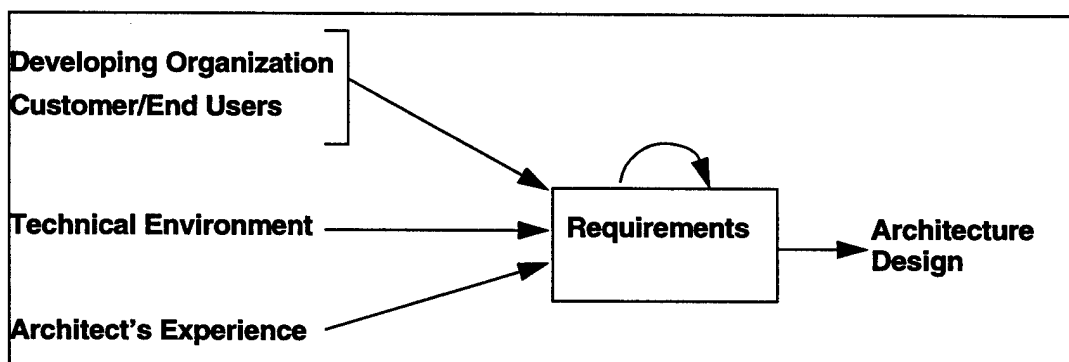


Figure 2-1: Eliciting the Architectural Requirements

The requirements can be subdivided into those related to the functions of the system and those related to the architecture. Since this report focuses on the architectural aspects of the design, we will not pursue organizing the functional requirements. Note, however, that the functional requirements tend to be very numerous, can be organized into different levels of abstraction, and are made concrete by the expression of use cases. We will assume that a relatively small list of classes of functionality is available for a subsequent step, so one of the organization's steps should be to generate this list. We will first discuss architectural requirements explicitly, and then address how they can be made concrete.

2.1 Architectural Requirements

It is typically possible to identify a small number of "architectural drivers" for a system to be designed. An architect experienced in a domain can look at a requirement that the architecture

must be suitable for a product line, for example, and identify four or five different architectural variation points that are important for the kind of systems that will be developed in the product line. An architect can look at a requirement that the system to be designed use the organization's database experts, and know that the system will have a database and will probably use the standard three-tier database architectural style. The first step in defining architectural requirements is to identify the architectural drivers.

The next step is to enumerate the architectural requirements. The requirements are an enumeration of the consequences of the architectural drivers and bring in other important architectural requirements. In the example where the driver is the desire for a product line, the requirements might be an enumeration of the architecture variation points and an enumeration of performance and reliability requirements. In the database example, the requirements would be an enumeration of variation (or fixed points) in the database management system being used and an enumeration of the types of modifications that might be made to the system and some performance requirements.

Architectural requirements derive from one of three sources: the quality goals for the system, the business goals for the system, or the business goals for the people who will work on the system. An example of the latter requirement comes from an organization that wishes to build a cadre of personnel familiar with C++ graphical user interface (GUI) development, where the requirement might be to use C++.

Architectural requirements are not as numerous as functional requirements; there should be a maximum of approximately 20 architectural requirements.

2.2 Quality Scenarios

The design process is based on the premise that the architectural requirements of a system or a collection of systems are as important as the behavioral requirements. Both types of requirements are made concrete in terms of scenarios or use cases. A variety of different types of scenarios are used. Normal use cases are used for the behavioral requirements for single products. Abstract scenarios, described in Section 2.2.1, are used for the behavioral requirements for product lines.

The quality-based architectural requirements, described below, are expressed via quality-specific scenarios, which are described in Section 2.2.2. The quality-specific scenarios are change scenarios in the case of modifiability, threat scenarios in the case of security, response-time scenarios in the case of performance, and error-handling or degradation scenarios in the case of reliability or availability.

However, even quality-specific scenarios have an impact on multiple qualities. For example, consider the following change scenario: "Change the system to add a cache to the client." It is

not only legitimate, it is mandatory, to ask about the effect of this change on performance, security, or reliability. Although one quality may be used to motivate a scenario, the impact of that scenario on other qualities needs to be considered.

Furthermore, the requirements come from many stakeholders. Why is this? No single stakeholder represents all the ways in which a system will be used. No single stakeholder will understand the future pressures that a system will have to withstand. Each of these concerns must be reflected by the scenarios that we collect.

We now have a complex problem however. We have multiple stakeholders, each of whom might have multiple scenarios of concern. They would rightly like to be reassured that the architecture satisfies all of these scenarios in an acceptable fashion. And some of these scenarios will have implications for multiple system qualities, such as maintainability, performance, security, modifiability, and availability.

We need to reflect these scenarios in the architectural structures that we document and the architectures that we build. In addition, we must be able to understand the impacts of the scenarios on the software architecture. We further need to trace the connections from a scenario to other scenarios, to the analytic models of the architecture that we construct, and to the architecture itself. As a consequence, understanding the architecture's satisfaction of the scenario depends on having a framework that helps us to ask the right questions of the architecture.

To use scenarios appropriately, and to ensure complete coverage of their implications, we typically consider three orthogonal dimensions, as shown in Figure 2-2.

	St 1	St 2	...	St N
Sc 1				
Sc 2				
Sc 3				
Sc 4				
...				
Sc N				

Figure 2-2: A Scenario-Elicitation Matrix

The entries in the matrix are the specific scenarios. This characterization allows us to manage the scenarios not only for specifying the requirements, but also for subsequently validating the architecture that is designed. The initial use of a quality-specific scenario might be considered during the design step, but the impact of that quality scenario on other qualities is also important during the analysis step.

2.2.1 Abstract Scenarios

In product lines, the architecture is the central reusable asset shared among the systems that are instances of the product line. Requirements for product lines are different from those traditionally collected for single-use systems. Consider the following example scenario: Database updates are propagated to clients within a bounded amount of time.

This scenario—propagating database updates to clients—embodies a requirement that can be specified only in an oblique fashion (i.e., a bounded amount of time). This lack of specificity is necessary because the scenario applies to a *family of systems*, rather than a single system, and so few assumptions can be made about the hardware platform, the environment in which the system is running, other tasks competing for resources, and so forth. Given the variability inherent in a family of systems, the stakeholders must resort to a *generic* requirement, represented by an *abstract scenario*. Traditional requirements of the form “the system shall do such and such” are frequently inappropriate for families of systems. They are replaced with abstract scenarios that represent entire classes of system-specific scenarios, with appropriate parameters. The use of abstract scenarios has several important implications:

- The architecture for a product line must identify infrastructure mechanisms for ensuring that this scenario can be met *in general*. For example, mechanisms to ensure that performance requirements are met may include performance monitoring, task scheduling (and potentially the support of multiple scheduling policies), as well as the ability to prioritize and preempt tasks. These mechanisms are necessary in a product-line architecture, because other forms of meeting performance deadlines that are appropriate in a single system (such as manual performance optimization) do not scale up to a family of systems.
- Tracing abstract scenarios from their original statement, through a specific version for a specific realization of the architecture, to a single realized system is much more complex than the equivalent tracing activity for a traditional single-use architecture. However, we need to maintain the scenario at this level of abstraction because it is only here that the requirement becomes a constraint on the architecture *as a reusable asset* rather than a system-specific goal that might be met, in a single-use architecture, through more traditional means such as code optimization or manual load balancing.

2.2.2 Quality-Specific Scenarios

The quality requirements for a product line are embodied in the quality scenarios. Qualities, in general, are too abstract for direct use in design. That is, every system is easy to modify for a certain class of changes and difficult to modify for others, so a statement requiring that a system "shall be modifiable," as is commonly included in requirements documents, is meaningless.

We make the quality requirements concrete by expressing them as quality-specific scenarios. As we mentioned above, quality-specific scenarios have implications for qualities other than the motivating one, but they are generated by the initial consideration of a specific quality. The qualities we typically consider are

- *modifiability*. In this case, the scenario is a change scenario. Since we are developing an architecture for a product line of systems, there is a natural variation already considered within the architecture, but many classes of changes will not be considered within the variation. For example, replacing the operating system or object request broker might not be considered within the abstract scenarios.
- *performance*. In this case, the scenario is a specification of the workload and the latency or throughput requirement. The form of the specification will depend on the type of system. In an interactive system, the form of the specification might be an abstract specification of the number of users and a deadline for response; in an embedded real-time system, the form of the specification might be a characterization of the input events and an associated deadline.
- *security*. In this case, the scenario is a particular type of threat and the response of the system to this threat. For example, an unauthorized user attempts to use the system; this should result in some sort of message. Another example is that an unauthorized user attempts to bypass the authentication system; this should also result in an intrusion alert being generated.
- *reliability*. In this case, the scenario is an exception or failure scenario together with the required system behavior. For example, if a particular type of failure occurs, then the system should be able to operate in a degraded mode.

2.3 Validation

The validation of a set of scenarios is accomplished through (1) scenario brainstorming with a wide group of stakeholders to ensure breadth and (2) mapping the scenarios onto the architecture requirements and then onto the business and human goals for the system to ensure coverage. Subsequently, the scenarios are used in the validation process for the architecture design.

3 Design the Architecture

An architect develops a design by making some collection of design decisions and then reasoning about these decisions through consideration of different architectural structures and views. Architectural requirements are used to motivate and justify design decisions, the different views are used to express information pertinent to achieve the quality goals, and the quality scenarios are used to reason about and validate the design decisions. Design decisions are frequently initiated by a knowledge of architectural styles [Shaw 96], design patterns [Buschmann 96], or the use of particular tools [sockets, remote procedure call (RPC), Common Object Request Broker Architecture (CORBA), etc]. Figure 3-1 provides an abstraction of the architecture design step.

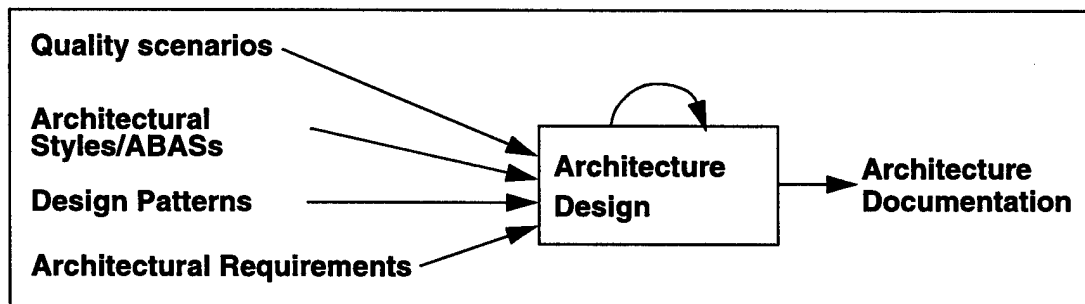


Figure 3-1: Architecture Design

Architecture design is an iterative process with some collection of decisions being made and reasoned about, then reconsidered and remade until closure on a design is reached. We discuss it by describing how the requirements feed into the consideration of the different structures and views and how the scenarios are used to reason about the various qualities. The concept that the architect is iterating through decisions and using multiple views simultaneously is different from the methods described by Kruchten [Kruchten 95], and Soni, Nord, and Hofmeister [Soni 95] who propose more of a sequential use of views. Their methods use one view to get a particular aspect of the architecture correct, at which point they move on to another view to get a different aspect correct. Our observation is that architects simultaneously use all of the views to refine their design, although they may defer serious consideration of some views until more of the architecture has been defined.

3.1 Architectural Structures and Views

A structure in software architecture is a set of like-type nodes connected by relations. When describing architectures, the nodes are commonly termed “components” and the relations are termed “connectors.” These components and connectors are annotated with other information that we call “properties.” Properties are used to differentiate between different types of components, different types of connectors, and to provide information useful for various architectural analyses (such as performance, security, or reliability analysis).

While there are many potentially interesting software structures, there are five canonical or foundational structures that together completely describe an architecture. Note that we have not yet defined precisely what we mean by the terms “view” or “structure.” Architectural structures are foundational representations of architectural information, representing the base artifacts that we design and code: classes, functions, objects, files, libraries, and so forth. Views are *derived* representations, arrived at by selecting a subset of a structure or by fusing information from multiple structures. These canonical structures may be elaborated in a variety of orders depending on the context of development. In Section 3.2, we provide a particular process that can be used to elaborate these structures. The five structures that we present here are derived from those presented by Kruchten [Kruchten 95].

The five canonical architectural structures are described below.

1. *Functional structure* is concerned with the decomposition of the functionality that the system needs to support. Components are functional (domain) entities, and the connectors are “uses” or passes-data-to. This structure is useful for understanding the interactions between entities in the problem space, for planning functionality, and for understanding the domain variability and hence the possibilities for creating a product line.
2. *Code structure* is concerned with the realization of the key code abstractions from which the system is built. Components can be packages, classes, objects, procedures, functions, methods, etc., all of which are vehicles for packaging functionality at various levels of abstraction. Relations include passes-control-to, passes-data-to, shares-data-with, calls, uses, is-an-instance-of, etc. This structure is crucial for understanding the maintainability, modifiability, reusability, and portability of the system. For example, how is the code broken down into subsystems and interfaces among them? How does the use of middleware affect this allocation?
3. *Concurrency structure* is concerned with logical concurrency. The components of this structure are units of concurrency that ultimately are refined to processes and threads. Relations include synchronizes-with, is-higher-priority-than, sends-data-to, can’t-run-without, can’t-run-with, etc. Properties relevant to this structure include priority, preemption, and execution time. This structure is key in understanding performance and is also important for reliability and security. Examples of the information contained in this view are the number, execution times, and priorities of the system’s threads and/or processes.

4. *Physical structure* is concerned with hardware including central processing units (CPUs), memory, buses, networks, and input/output (I/O) devices. Properties relevant to this structure include availability, capacity, and bandwidth. For example, is there any provision for redundant hardware and networks?
5. *Developmental structure* is concerned with files and directories. This structure is important for managing and ensuring administrative control of the system as it grows and is fleshed out, including the division of work into teams and configuration management.

One reason why architects use multiple views simultaneously is that each view exposes some information and hides other information. Thus, for example, when reasoning about the functions of the system and the modifiability of the system, distribution issues are not the primary concern. The code structure abstracts away distribution issues. On the other hand, when reasoning about performance, distribution is primary and the function of the system becomes secondary, so a concurrency structure is more appropriate.

Table 3-1 summarizes some of the relationships between architectural structures and the qualities for which they support reasoning.

Quality	Useful Structures
Performance	Concurrency, Physical
Security	Concurrency, Code
Reliability/ Availability	Concurrency, Physical
Modifiability/Maintainability	Functional, Code, Development

Table 3-1: Views Useful for Different Qualities

3.2 A Development Process

The development context will determine the appropriate development process. If the system being developed can derive much from existing systems, then a development process that assumes the views for the existing system is appropriate. In this section, we present a process for a green-field development. That is, we assume that the system being developed is not constrained to use pre-existing components. The design may (and probably will) assume the use of available components, so we assume that the use of particular components is not a *constraint* to the design. Any components that are to be used are the choice of the architect and are not prespecified. Furthermore, at the initial design stages, the development and code structures are not specified since the units of design have not yet been decided.

We begin by assuming that we have a list of architectural requirements and a list of classes of functionality derived from the functional requirements. The goal of the first step is to develop a list of candidate subsystems. All of the classes of functionality that are derived from the functional requirements are automatically candidate subsystems. We derive the other candidate subsystems from the architectural requirements.

For each architectural requirement, we enumerate possible architecture choices that would satisfy that requirement. For example, if the requirement is to allow for the change of operating systems, then an architectural choice to satisfy that requirement is to have a virtual operating system adaptor. Some architectural requirements may have multiple possible choices for satisfaction, while others may have a single choice. This enumeration comes from a consideration of design patterns, architectural styles, and the architect's experience.

From this list of possible choices, a selection is made so that all of the architecture requirements are potentially satisfied, the list of choices is consistent, and the list is minimized. By minimizing the list of choices, we mean that if one choice will satisfy two requirements (even if less than optimally), then that choice should be preferred over two different choices. Fewer choices mean fewer components, less work in implementation and maintenance, and, in general, fewer possibilities for error. Each of the choices is added to the list of candidate subsystems.

The next step in the process is to choose the subsystems. Each candidate subsystem will be categorized as an actual subsystem, a component in a larger subsystem, or expressible as a pattern to be used by the actual subsystems. We then record the actual subsystems in the functional structure.

Once a list of actual subsystems is generated, the next step is to populate the concurrency structure. This structure is populated by reasoning about units of distribution and units of parallelism with respect to the subsystems. Each subsystem may be distributed over several physical nodes, in which case the units of distribution are identified as components belonging to that subsystem. Units of parallelism are identified by reasoning about threads and synchronization of threads within the subsystem. The threads are "virtual threads" in that they identify elements of parallelism if all of the subsystems were to reside on a single processor. When the physical structure is considered, it is possible to identify the places where the virtual threads turn into physical threads and the necessary network messages derived from this transformation.

At the end of this design step, the subsystems and their concurrency behavior have been identified. This step is then validated and the subsystems are refined, the concurrency behavior again identified, other structures populated, and so forth. Of course, the validation step may cause decisions to be revisited and so the actual process is highly iterative between decisions and validation.

3.3 Validation

The quality scenarios are used as the primary validation mechanism. The proposed structures are examined via the quality scenarios to see if the scenarios are achievable at the current level of design. If they are, then the next refinement of the design can proceed; if not, then the design at the current level of refinement must be reconsidered.

Each of the iterations described in this report is actually performing a mini-architecture analysis. The scenario-based reasoning can be used for design, as we have described, or in an evaluative setting. The scenarios structure the analysis, describing and operationalizing the performance threads, operational failures, security threats, and anticipated changes to the system. The mapping of these scenarios onto the appropriate architecture structures and views tells us *what* to analyze. For example, one typically does not want to analyze system performance as a whole—it is too vague and there are far too many things that could be measured. We typically want to predict average throughput or worst-case response time or some similar measure, as derived from and motivated by particular usage scenarios.

Once the architecture has been designed, it is useful to have an architectural evaluation carried out by a group external to the developers. An external analysis makes the results of the design process visible to management and forces the documentation of the architecture.

4 Document the Architecture

An architecture's documentation is designed to support the needs of the programmers and analysts. It can be a tremendous vehicle for enhancing communication among the stakeholders and for eliciting architectural requirements from them. Creating and maintaining the architectural documentation, as graphically depicted in Figure 4-1, is a critical success factor in a long-lived software architecture.

Most architectural constructs are abstract, consisting of groupings of components that (ideally) have conceptual integrity. For example, the concept of a *layer* does not exist in any programming language, and one of the major uses of software architecture is as a communication device among the system's stakeholders. So, we claim that an architecture for a software system does not really exist except in its documentation [Kazman 99a]. Thus the completeness and quality of the documentation is a crucial factor in the success of the architecture. An architecture should be documented, at a minimum, according to the structures detailed in Section 3.1.

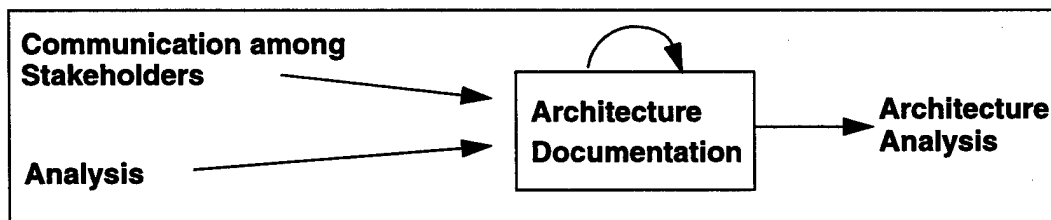


Figure 4-1: Architecture Documentation

Below are our major recommendations for the documentation of a software architecture.

1. The documentation should be complete and navigable. That is, a skilled software engineer with some domain knowledge but no prior knowledge of this architecture should be able to read the documentation and navigate through it. There should be an obvious starting point, portraying the system as a collection of interconnected subsystems. The subsystems should be named, their responsibilities and functionality identified, and the nature of their

interconnections identified. Pointers there should direct the reader to more detailed documentation of subsystems, and then components. At every stage, the nature of the connections among the parts shown should be clearly identified: sub-class, data flow, control flow, parallel process, etc.

The documentation, at this point, is not likely to be complete in terms of every aspect addressed in detail. However, the concepts, top-level structure, and important lower-level details should be complete. It may still change, but it should be a complete design with respect to the above-mentioned areas. Areas of incompleteness should be clearly identified so that a user knows what information to expect when the documentation is finished.

This means that the documentation must be presented as a “big-picture” reference architecture that “wires together” the various subsystems, as well as showing how subsystems are wired together internally.

2. The infrastructure—the architecture’s set of mechanisms for communication and coordination—must be documented as an integral part of the architecture. If the software architecture is for a product line, the infrastructure is the one constant (any of the functionality that lives on top of the infrastructure could change in the future) and provides the means for predicting, measuring, and ensuring the system’s quality attributes once the specific functionality is determined. The infrastructure is also reused by all of the subsystems, including those that are unique to the target system, in the sense that these subsystems comply with, and are derivable from, the abstractions in the infrastructure.

Because the infrastructure is reused by all the subsystems, it is not enough to know about logical data and control flow such as would be found in message sequence charts (that do not include the infrastructure as part of their sequencing). We need to see how these logical relations are realized through use of the infrastructure. For example, a message sequence chart might indicate that *A* sends a message to *B*. We need to know how that message is transmitted: via RPC, event/notification, hypertext transfer protocol (HTTP), etc., and which infrastructure components are responsible for effecting the transaction.

3. As part of the architecture documentation, a number of use cases must be mapped onto the architectural representation to a sufficient detail that a software engineer can understand how the system will implement the functionality (in terms of processing, data flow, and control flow through the infrastructure components). These scenarios represent the major uses of the system and will demonstrate the leverage obtained from a small number of infrastructure components satisfying a large number of application needs.
4. The architecture and its use in target systems must be bound by a set of constraints on mechanisms for communications, management of resources for data distribution, time management, and other infrastructure services. There should be a prespecified application framework for the infrastructure and a prespecified minimal number of communication mechanisms. Any part of the architecture that does not use either the application framework or the specified communication mechanism should include a justification pointing to

an analysis of performance, fault tolerance, maintainability, security, or some other quality that argues against the use of the standard mechanism. The point here is that without a consistent set of design constraints, it is difficult to make measurements of these system qualities to determine if an alternative offers the required improvements.

5. *All* the documentation should be made publicly available to all stakeholders.

The development of architectural structures and views, and the mapping of both use cases and quality scenarios onto the various views, involves a tremendous amount of detailed information. Tool support is vital for managing this information, keeping the various views consistent, and keeping the documentation up to date. In Chapter 5, we describe considerations with respect to tool support.

5 Analyze the Architecture

The design, documentation, and mini-architecture analyses are iterated until the major architectural decisions have been made. At that point, there should be a major architecture evaluation involving external reviewers. The purpose of the evaluation is to analyze the architecture to identify potential risks and to verify that the quality requirements have been addressed in the design. The reason for having external reviewers is to ensure that the design is examined with impartiality and to create credibility for the results of the evaluation with the management of the organization that is building the system. Figure 5-1 shows an abstraction of the analysis process.

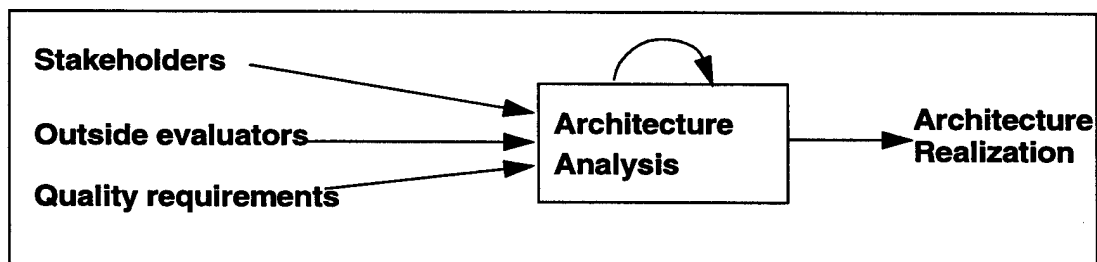


Figure 5-1: Architecture Analysis

A variety of different types and styles of architecture evaluations exist. Some organizations such as AT&T/Lucent have been doing architecture evaluations for 10 years, and evaluations are increasingly being required for large, risky systems. Architecture evaluations provide the benefit of forcing documentation of the architecture (for those systems not following the process we give here), early detection of risk areas, and improved architectures. We discuss architecture evaluations in general, and then we discuss a type of evaluation designed to evaluate for the particular qualities of modifiability, performance, reliability, and security.

5.1 Architecture Evaluations

When performing an architecture evaluation, there are two issues that are of importance: who participates in the evaluation and review how the evaluation is performed. We discuss these issues in Section 5.1.1 and Section 5.1.2.

5.1.1 Participants

Clearly, the architect must participate in any architecture evaluation. The architect should be supported by as many members of the design team as appropriate. For a large system, participants should include the lead designers of the major subsystems, but possibly also the testers, maintainers, integrators, system administrators, end users, customers, and managers. In short, anyone who has a stake in the system's success should be a potential attendee.

The evaluation team is another set of participants that must be planned. The evaluation team should be external to the development team for impartiality and a fresh perspective. It must have people expert in architectural issues and also people expert in the domain of the system being reviewed.

Finally, the evaluation should include representatives of the stakeholders. Some of the issues that arise at a review involve process and requirement issues. The stakeholders' input is vital to determining appropriate responses to these issues; the evaluation team cannot hope to understand all of the details of the application domain and all of the business issues. The team relies on the stakeholders to provide this information. The stakeholders also bring the perspective of possible changes in either requirements or the environment that might affect the architecture. Questions such as "What is the impact of a 20% budget cut?" are generated by the appropriate stakeholders.

5.1.2 Evaluation Techniques

Evaluation techniques are either question based or quantitative analysis based. Question based reviews operate from a list of questions or scenarios either generated during the evaluation or based on prior experience. An organization may have a list of questions that pertain to a particular type of system such as "How are you handling a fault caused by a divide by 0?", "How do you do a live switch between copies of the database?", or "How will the system respond under the following peak operator load assumptions?" These questions should be provided to the development team during the design process since they represent the accumulated organizational wisdom with respect to the problems that arise during the development of a particular type of system. Other questions are generated during the evaluation by the stakeholders and are specific to the system being developed. An organization can accumulate the questions developed during reviews and use them as the basis for the questionnaires provided prior to the review.

The second type of evaluation technique is quantitative based and is based on the analysis of a particular attribute and the existence of models for these attributes. Performance, for example, has been well studied; there is a collection of information that will enable the building of a performance model, and knowing the model and its requirements enables eliciting the information necessary to instantiate the model. Other attributes that have models useful in analysis are reliability, security, liveness, and reachability.

5.2 Architecture Tradeoff Analysis Method (ATAM)

In this section we describe a particular architecture evaluation method, called the Architecture Tradeoff Analysis Method (ATAM). The problem with evaluating an architecture is the reverse of the design problem. The evaluators must understand the architecture, recognize the architectural parameters that were used, know the implications of these architectural parameters with respect to quality attributes, and then compare these implications to the requirements for the system. A further problem is that the documentation for both the requirements and the architecture are often incomplete with respect to the needs of an evaluation.

Since an architecture evaluation is intended to be performed on an architecture and not on an existing system, the output is, inherently, imprecise. The results of an architecture evaluation, then, are used as an indication to the architect of problem areas in the architecture and as a risk reduction mechanism by management. The problem areas identified, in ATAM are called "sensitivity points" and "tradeoff points." A sensitivity point is a collection of components in the architecture that are critical for the achievement of a particular quality attribute. A tradeoff point is a sensitivity point that is critical for the achievement of multiple attributes.

Given the problems enumerated, ATAM uses the following elements to determine the sensitivity and tradeoff points:

- Quality scenarios (both expected and unexpected) are used as a manifestation of the quality attribute requirements.
- Stakeholders are used as the generators of the quality scenarios.
- Quality scenarios and use cases are used as an elicitation technique for the architecture.
- The architect is used as the interpreter of the quality scenarios and the use cases.
- Quality attribute taxonomies are used to provide the evaluators with a catalog of architectural parameters and the appropriate stimuli.

An ATAM is typically planned to take three full days where each day consists, in some measure, of

- scenario elicitation
- architecture elicitation
- mapping of scenarios onto the architecture representation
- analysis

During days 1 and 2, there is more emphasis on the early steps of the method (scenario elicitation, architecture elicitation, and scenario mapping). During days 2 and 3, there is more emphasis on the later steps of the method (model building and analysis, sensitivity point identification, tradeoff point identification). Graphically, we see the relationship of the three activities with respect to time as shown in Figure 5-2.

In Figure 5-2, the width of the polygon at each activity shows the amount of anticipated activity within that activity at that time (i.e. on day 1 we expect relatively little analysis to take place, and on day 3 we expect most of the activity to be taken up with scenario mapping and analysis).

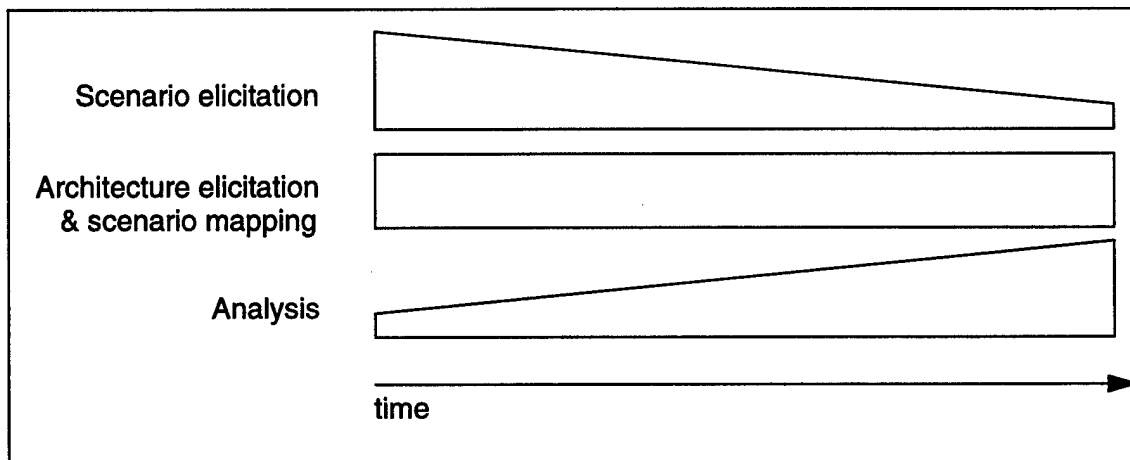


Figure 5-2: The Activities of ATAM and Their Relative Importance Over Time

5.3 The Steps of the ATAM

The steps of the ATAM are presented in Section 5.3.1 and Section 5.3.2. These steps are divided into three days of activities but the division is not a hard-and-fast one. Sometimes there must be dynamic modifications to the schedule to accommodate the availability of personnel or architectural information.

The ATAM is not a waterfall process. There will be times when an analyst will return briefly to an earlier step, jump forward to a later step, or iterate among steps, as the need dictates. The importance of the steps is to clearly delineate the activities involved in ATAM along with the outputs of these activities. What is more important than the particulars of the schedule is the set of dependencies among the outputs of the steps, as indicated in Figure 5-3.

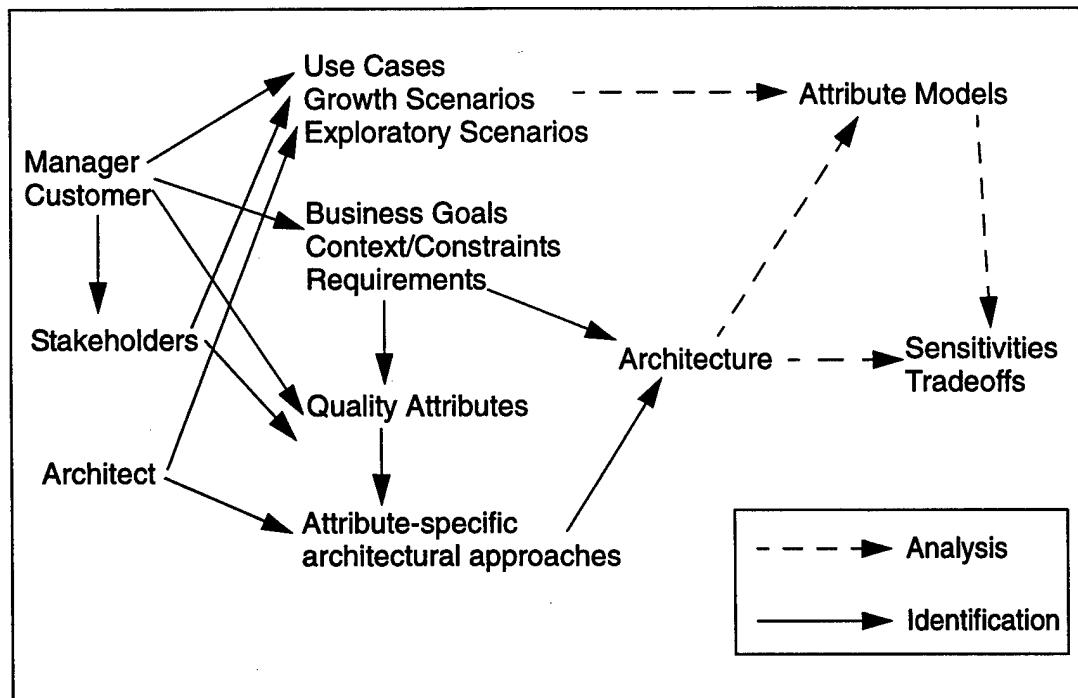


Figure 5-3: Dependencies Among ATAM Outputs

5.3.1 Day 1 Activities

On day 1, the ATAM team meets the team being evaluated, perhaps for the first time. This meeting has two concerns: the organization of the rest of the analysis activities and information collection. Organizationally, the manager of the team being evaluated needs to make sure that the right people attend the meetings, that the people are prepared, and that they come with the right attitude (i.e., a spirit of non-adversarial teamwork). The information-collection aspect of day 1 is meant to ensure that the architecture can truly be evaluated, meaning that it is represented in sufficient detail. Also, some initial scenario collection and analysis may be done on day 1, as a way of understanding the architecture, understanding what information needs to be collected and represented, and understanding what it means to generate scenarios. The following steps take place on day 1.

Step 1 - Lead evaluator presents ATAM. In this step the ATAM is presented to the assembled stakeholders (typically managers, customer representatives, and architects). This step sets the context and expectations for the remainder of the activities.

Step 2 - Manager/Customer presents system overview. The system to be evaluated needs to be understood by all participants in the evaluation. This understanding has several facets. The system itself must be presented, initially at a high level of abstraction. This presentation typically describes the system's major functions, requirements, constraints, business

goals, and context. Also, when describing the system, all of the system's stakeholders need to be identified, so that they can be invited to contribute to later steps of the ATAM.

Step 3 - Architect describes important attribute-specific requirements. The architect needs to explain the driving architectural requirements, in terms of performance, modifiability, security, reliability, and so forth. These attribute-specific requirements shape the architecture, much more than its functional requirements.

Step 4 - Architect presents architecture. The architecture is presented in as much detail as is currently documented. This is an important step, as the amount of architectural information available and documented will directly affect the analysis that is possible, and its quality. Frequently the evaluator will have to specify additional architectural information that must be collected and documented before proceeding to days 2 and 3 of the ATAM.

Step 5 - Architect presents attribute-specific architectural approaches. For each of the attribute-specific requirements presented in Step 3, the architect presents the architectural approach that is intended to meet the requirement. These approaches will be important to analyze in days 2 and 3, since they represent the major architectural decisions intended to meet the major quality-attribute requirements.

Step 6 - Evaluator elicits "seed" use cases and scenarios. Scenarios are the motor that drives the ATAM. Generating a set of "seed" scenarios (and use cases) has proven to greatly facilitate discussion and brainstorming during days 2 and 3, when greater numbers of stakeholders are gathered to participate in the ATAM. These "seed" scenarios serve as models to the stakeholders by defining the nature and scope of appropriate scenarios for evaluation.

Step 7 - Architect maps seed use cases onto architecture. In this step, the architect walks through a selected set of the seed use cases and scenarios, showing how each affects the architecture (e.g., for modifiability) and how the architecture responds to it (e.g., for quality attributes such as performance, security, and availability).

Step 8 - Evaluators build initial skeleton analysis. As a result of presenting attribute-specific architectural approaches and mapping seed scenarios and use cases on to the architecture, the evaluators should now be able to build initial models of each important quality attribute. These will likely not be formal models at this point, but rather informal models. For example, informal models will include the known strengths and weaknesses of a particular approach and sets of questions that help identify these strengths and weaknesses. Such models will guide the evaluators in the later stages of the ATAM in probing for more precise and detailed information.

Step 9 - Evaluators determine action items. A number of needs have typically become evident at this point in the evaluation. For example, as stated above, the architectural documentation is often insufficient to support a complete evaluation and this must be remedied as

an action item. A set of stakeholders must be contacted and meeting dates for days 2 and 3 must be determined. In addition, any action item may become a go/no-go decision for the continuation of the ATAM.

5.3.2 Day 2 Activities

During day 2, the main analysis activities begin. At this point, it is assumed that the architecture has been documented in sufficient detail to support analysis, that the appropriate stakeholders have been gathered and have been given advanced reading materials so that they know what to expect from the ATAM, and that the seed scenarios have been collected and distributed to the stakeholders. The following steps take place on day 2.

Step 1 - Lead evaluator presents ATAM. Since there will be a different set of stakeholders attending day 2, and since a number of days or weeks may have transpired between days 1 and 2, it is useful to recap the steps of the ATAM, so that all attendees have the same understanding and expectations of the day's activities.

Step 2 - Evaluators brainstorm scenarios/use cases with stakeholders. The stakeholders now undertake two related activities: brainstorming use cases (representing the ways in which the stakeholders expect the system to be used) and scenarios (representing the ways in which the stakeholders expect the system to change in the future). The scenarios are further subdivided into two categories: *growth scenarios* and *exploratory scenarios*. Growth scenarios represent ways in which the architecture is expected to accommodate growth and change: expected modifications, changes in performance or availability, porting to other platforms, integration with other software, and so forth. Exploratory scenarios, on the other hand, represent extreme forms of growth: ways in which the architecture might be stressed by changes, dramatic new performance or availability requirements (order of magnitude changes, for example), major changes in the infrastructure or mission of the system, and so forth. Growth scenarios are a way of showing the strengths and weaknesses of the architecture with respect to anticipated forces on the system. Exploratory scenarios are an attempt to find sensitivity points and tradeoff points. Identifying these points helps us assess the limits of the system with respect to the models of quality attributes that we build.

The seed scenarios created in day 1 help facilitate this step by providing stakeholders with examples of relevant scenarios.

Step 3 - Evaluators prioritize scenarios/use cases with stakeholders. Once the use cases and scenarios have been collected, they must be prioritized. We prioritize each category of use case/scenario separately. We typically do this via a voting procedure where each stakeholder is allocated a number of votes equal to 30% of the number of scenarios within the category, rounded up. So, for instance, if there were 18 use cases collected, each stakeholder would be given 6 votes. These votes can be allocated in any way that the stakeholder sees fit:

all 6 votes allocated to 1 scenario, 2 votes to each of 3 scenarios, 1 vote to each of 6 scenarios, etc. This can be an open or a secret balloting procedure. Once the votes have been made, they are tallied, and the use cases and scenarios are prioritized by category. A cutoff is typically made that separates the high-priority use cases from the lower ones, and only the high-priority ones are considered in future evaluation steps. For example, a team might only consider the top 5 use cases.

Step 4 - Architect maps important use cases onto architecture. The architect, considering each high-priority use case in turn, maps the use cases onto the architecture. In mapping these use cases, the architect walks through the actions that the use case initiates, highlighting the areas of the architecture that participate in realizing the use case. This information can be documented as a set of components and connectors, but it should also be documented with a pictorial representation that shows the flow of responsibility among the components and connectors. The choice of which documentation vehicle to use depends in part on the granularity of information required, the perceived risk implied by the scenario, and the stage of the architecture's development. For a mature project and/or high-risk project, it may be possible and important to trace through behavior diagrams. For a less mature or less risky project, use case maps might be appropriate. The evaluators can use this walkthrough of the use case to ask attribute-specific questions (such as performance or reliability questions) and to annotate the architectural representation. In doing so, the evaluators will record a set of issues, sensitivity points, and trade-off points.

Step 5 - Architect maps important growth scenarios onto architecture. As with step 4, the architect considers each high-priority growth scenario in turn and maps each of these onto the architecture. Since growth scenarios imply a change to the system, the architect identifies all changed components and connectors. In addition to understanding and documenting the modifications to the system, the evaluators probe for the issues, sensitivities, and tradeoffs uncovered by this scenario since they affect quality attributes such as performance, security, and reliability.

Step 6 - Evaluators build skeleton analyses of each quality attribute. Using a set of organization-wide screening questions and qualitative analysis heuristics, the evaluators are now in a position to begin building more complete models of each quality attribute under scrutiny. Building a model of even a single quality attribute can be a daunting task in a complex system; there is simply too much information to assimilate. Our solution to this information overload problem is to use screening questions and qualitative analysis heuristics. The screening questions serve to limit the portion of the architecture under scrutiny. The qualitative analysis heuristics suggest questions for the evaluator to ask of the architect that help in identifying common architectural problems. If these questions do uncover potential problems, then the analyst can begin to build a more comprehensive model of the quality-attribute aspect under scrutiny.

5.3.3 Day 3 Activities

Step 1 - Brainstorm exploratory scenarios with selected stakeholders. As was done on day 2, day 3 begins with the brainstorming of scenarios, in this case exploratory scenarios. The difference here is that the stakeholder group on the third day has now been pared down to just the lead architects and key developers. These scenarios, as with all scenario brainstorming, are examined in light of the quality attributes that they cover (to ensure that all important attributes are represented fairly) and the stakeholders whose concerns they address.

Step 2 - Prioritize exploratory scenarios. As was done on day 2, we then prioritize the newly captured exploratory scenarios. This setting of priorities can proceed exactly as it did on day 2 (if the group of stakeholders is of moderate size), or it can be done informally if the stakeholder group is small (say, fewer than six).

Step 3 - Map important exploratory scenarios onto relevant architectural views. The highest priority scenarios are now mapped onto the relevant architectural view or views, as on day 2. Once again, attribute-specific questions will be asked during this mapping process, and the answers will be recorded in the appropriate documentation template.

Step 4 - Build the analyses using screening questions and qualitative analysis heuristics. The skeleton analyses that were created on day 2 are further fleshed out during this stage. Once again, we need to use the screening questions and qualitative analysis heuristics to limit the scope of the investigation. However, building on the models from day 2, we can begin to explore areas that appear to be potential locales of issues, sensitivities, and global tradeoffs.

Step 5 - Debriefing. By the end of three days of analysis, the analysts have amassed a substantial amount of information on the system under scrutiny. The analysts have also begun to form models of the important quality attributes.

It should be noted that this has been just a brief sketch of the steps of the ATAM process. More details and examples can be found in [Kazman 99a].

6 Realize the Architecture

When turning an architecture into code, all of the usual software engineering and project management considerations must be taken into account: doing detailed design, implementation, testing, configuration management, and so forth. One thing that is specific, however, to architecture-based development is the structure of the organization. In particular, the organizational structure of the development team must be easily mapped onto the software architecture, and vice versa.

Conway eloquently expressed the necessity for this mapping over 30 years ago [Conway 68]:

Take any two nodes x and y of the system. Either they are joined by a branch or they are not. (That is, either they communicate with each other in some way meaningful to the operation of the system or they do not.) If there is a branch, then the two (not necessarily distinct) design groups X and Y which designed the two nodes must have negotiated and agreed upon an interface specification to permit communication between the two corresponding nodes of the design organization. If, on the other hand, there is no branch between x and y , then the subsystems do not communicate with each other, there was nothing for the two corresponding design groups to negotiate, and therefore there is no branch between X and Y .

Conway was describing how to assess organizational structure (at least in terms of the presence or absence of communication paths) from the structure of a system. However, the relationship between organizational structure and system structure is bidirectional, and necessarily so.

The impact of an architecture upon the development organizational structure is clear. Once an architecture for the system under construction has been agreed upon, teams are allocated to work on the major components and a work breakdown structure is created that reflects those teams. Each team then creates its own internal work practices (or a system-wide set of practices is adopted). For large systems, the teams may belong to different subcontractors. The work practices will include items such as bulletin boards and Web pages for communication, file naming conventions for files, and the version control system that is adopted. All of these may be different from group to group, again especially for large systems. Furthermore, quality assurance and testing procedures will be set up for each group and each group will need to establish liaisons and coordinate with the other groups.

So, the teams within an organization work on components. Within the team there needs to be high-bandwidth communications: a large amount of information in the form of detailed design decisions is being shared constantly. Between teams, low-bandwidth communications are sufficient. (This is, of course, assuming that the system has been designed with appropriate separation of concerns.) Systems of high complexity result when these design criteria are not met. In fact, team structure and controlling team interactions often turn out to be the largest single factor affecting a large project's success. If interactions between the teams need to be complex, it means either that the interactions between the components they are creating are needlessly complex, or that the requirements for those components were not sufficiently "hardened" before development began. In this case, there is a need for high-bandwidth connections *between* teams, not just within teams, requiring substantial negotiations and often rework of components and their interfaces. Teams, like software systems, should strive for loose coupling and high cohesion.

7 Maintain the Architecture

We have now discussed how to design, document, analyze, and realize a software architecture, as well as its relationship to quality attributes. However, just having a software architecture is not the same as having an architecture that is well documented, well disseminated, or well maintained. If any of these activities are not done, then the architecture will inevitably drift from its original precepts. This is a risk; if the architecture drifts in multiple ways (because of changes by multiple developers) that are not mutually consistent, or which do not follow the original rationale for design decisions, then the achievement of quality attributes that had been so carefully designed and analyzed in the original system will be compromised. So, how can we ensure that the architecture of the as-designed system and the architecture of the as-built and as-maintained system remain congruent?

Manually assessing the conformance of an architecture to its design is a dreary and error-prone task. So, one technique that has been gaining popularity over the past five years is to use tools to extract the architecture of the as-built system and check it for *conformance* to the as-designed system.

However, this technique, even with tool support, is not straightforward for several reasons:

- Software architectures are seldom documented in practice.
- When they are documented, they are often not maintained.
- When they are documented, the documentation is often ambiguous.

This last point is worth some attention. Many architectural constructs have no realization in the development artifacts that programmers actually create and maintain. Nowhere in the code and header files of a typical source repository will we find a layer, a subsystem, a functional grouping, or a class category. These concepts typically exist only within the minds of the architect and a select group of programmers, and their mapping to development artifacts is unclear. Hence, architectural reconstruction typically has an *interpretive* aspect, where the architect associates certain naming conventions, file or directory structures, or structural constraints with an architectural construct. For example, a layer might be realized as any function that directly accesses the database. A subsystem might be defined by all of the files in the *IO* directory, or by all subclasses of the *PrimitiveOp* class. Yet these abstractions are useful and we want to be able to support their existence and enforce their continued existence throughout a system's lifetime.

There are other reasons for wanting to extract the software architecture of an existing system than simply for redocumentation:

- An organization may want to reengineer an existing system so it needs to know what assets it is currently working with to plan the reengineering effort, or to decide that the existing assets are not worth reusing.
- An organization may want to mine its existing assets to form a reuse library or the core of a product line, so it needs to know what dependencies on those assets exist in the system.
- An organization may want to analyze its existing system with respect to its future prospects, growth potential, suitability for integration with other systems, or scalability. For each of these analyses, an accurate representation of the architecture is a crucial prerequisite.

Tools, such as the Dali workbench [Kazman 99b] have been developed to aid an analyst in extracting information from an existing system. The extracted information can be wide ranging in scope: its code, build files, execution traces, results of instrumentation, file structure, etc. The analyst then needs to work with the architect defining the patterns that describe such mappings. Typically, this is an iterative, interpretive process. The patterns are defined, applied to the extracted artifacts, and viewed by the architect, potentially resulting in a redefinition of the patterns. Once such patterns have been defined and suitably refined, they become the set of rules that *define* the architecture and against which the as-built architecture can be assessed for conformance.

These rules are then typically applied to the extracted artifacts, and any remaining anomalies can be manually or automatically noted. Based upon the results of this extraction and conformance activity, three possible outcomes may result: the documentation might be updated to reflect the reality of the existing code base; the code base may be brought into conformance with the architectural rules that have been explicitly defined (perhaps for the first time); or the anomalies might simply be recorded as anomalies with no further action for the time being. Whichever of these options is chosen, the result is increased understanding and hence intellectual control over the system and its future uses. This activity is thus a crucial step in a complete view of the architectural life cycle: the architecture must be documented and maintained just as any other system asset is maintained.

A complete discussion of architectural reverse engineering and conformance testing is beyond the scope of this report. For more information, the interested reader is encouraged to investigate [Kazman 99b] and the references cited therein.

8 Conclusions

In this report, we have discussed a set of steps for doing architecture-based development. This process differs from traditional development in that it concentrates on driving design and maintenance from the perspective of a software architecture. The motivation for this change of focus is that a software architecture is the placeholder for system qualities such as performance, modifiability, security, and reliability. The architecture not only allows designers to maintain intellectual control over a large, complex system but also affects the development process itself, suggesting (even dictating) the assignment of work to teams, integration plans, testing plans, configuration management, and documentation. In short, the architecture is a blueprint for all activities in the software development life-cycle. As such, the development process surrounding software architecture needs to be scrutinized.

References

- [Abowd 96] Abowd, G.; Bass, L.; Clements, P.; Kazman, R.; Northrop, L.; Zaremski, A. *Recommended Best Industrial Practice for Software Architecture Evaluation* (CMU/SEI/96-TR-025, ADA320786). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.
<http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.025.html>
- [Barbacci 97] Barbacci, M.; Carrière, J.; Kazman, R.; Klein, M.; Lipson, H.; Longstaff, T.; Weinstock, C. *Toward Architecture Tradeoff Analysis: Managing Attribute Conflicts and Interactions* (CMU/SEI-97-TR-029, ADA343692). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1997. <http://www.sei.cmu.edu/publications/documents/97.reports/97tr029/97tr029abstract.html>
- [Bass 98] Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*. Addison-Wesley, 1998.
- [Buschmann 96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-Oriented Software Architecture*. Wiley, 1996.
- [Conway 68] Conway, M. "How do Committees Invent?" *Datamation* (April 1968): 28-31.
- [Garlan 97] Garlan, D.; Monroe, R.; Wile, D. "ACME: An Architecture Description Interchange Language," 169-183. *Proceedings of CASCON '97*. Toronto, ON, Nov. 1997.
- [Jacobson 92] Jacobson, I.; Christerson, M.; Jonsson, P.; Overgaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

- [Kazman 99a]** Kazman, R.; Barbacci, M.; Klein, M.; Carrière, J. "Experience with Performing Architecture Tradeoff Analysis." *Proceedings of ICSE 21*. Los Angeles, CA, May 1999 (to appear).
- [Kazman 99b]** Kazman, R.; Carrière, J. "Playing Detective: Reconstructing Software Architecture from Available Evidence." *Journal of Automated Software Engineering* 6:2 (Apr. 1999): 107-138.
- [Kazman 99c]** Kazman, R.; Abowd, G.; Bass, L.; Clements, P. "Scenario-Based Analysis of Software Architecture." *IEEE Software* (Nov. 1996): 47-55.
- [Kazman 96]** Kazman, R. "Tool Support for Architecture Analysis and Design," 94-97. *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*. San Francisco, California, Oct 1996.
- [Kazman 93]** Klein, M.; Ralya, T.; Pollak, B.; Obenza, R.; Gonzales Harbour, M. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic, 1993. Ordering Information available at <http://www.sei.cmu.edu/topics/products/publications/rma.hndbk.html>
- [Kruchten 95]** Kruchten, P. "The 4+1 View Model of Software Architecture." *IEEE Software* (Nov. 1995): 42-50.
- [McCall 94]** McCall, J. "Quality Factors," 958-969. *Encyclopedia of Software Engineering* Vol 2. Wiley: New York, 1994.
- [Shaw 96]** Shaw, M.; Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [Smith 93]** Smith, C.; Williams, L. "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives." *IEEE Transactions on Software Engineering* 19, 7 (1993): 720-741.
- [Soni 95]** Soni, K; Nord, R.; Hofmeister, C. "Software Architecture in Industrial Applications." *Proceedings of the International Conference on Software Engineering*. Seattle, 1995.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<p>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</p>				
1. AGENCY USE ONLY (leave blank)		2. REPORT DATE April 1999		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Architecture-Based Development			5. FUNDING NUMBERS C — F19628-95-C-0003	
6. AUTHOR(S) Len Bass, Rick Kazman				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-99-TR-007	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-99-007	
11. SUPPLEMENTARY NOTES				
12.a DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12.b DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) This report presents a description of architecture-centric system development. In an architecture-centric process, a set of architecture requirements is developed in addition to functional requirements. This report describes the source of these architecture requirements and how they are elaborated into a design. In addition to design, the documentation, evaluation, realization, and maintenance of an architecture are also described.				
14. SUBJECT TERMS **KEY WORDS			15. NUMBER OF PAGES 36	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	